CHAPTER

THREE

CONCURRENCY

The last chapter described linguistic mechanisms for specifying concurrency. However, naively written concurrent programs produce unexpected results. In this chapter, we introduce the problem of resource conflict, describe synchronization mechanisms to resolve that conflict, and present several problems that illustrate synchronization and control issues.

3-1 RESOURCE CONFLICT

Unstructured concurrent computation produces unexpected results. For example, even the simplest of programming rules, "after assigning a value to a variable, that variable has that value (until that program makes some other assignment)," is not true of concurrent systems. We illustrate this difficulty with a procedure to add a deposit to "bank account" (variable) MyAccount:

```
procedure deposit (var amount: integer);
begin
    MyAccount := MyAccount + amount;
end
```

Suppose we try to make two deposits at the same time, as might arise from two simultaneous transactions in different branches of the bank. That is, we imagine that we have $1000 in MyAccount, and we execute

```
parbegin
    deposit(100);
```

```
    deposit (50)
parend
```

This is equivalent to executing, in parallel, the statements

```
parbegin
    MyAccount := MyAccount + 100;
    MyAccount := MyAccount + 50
parend
```

But neither of these assignment statements is itself primitive. Instead, incrementing an account is a series of operations such as

```
reg       := MyAccount;
reg       := reg + deposit;
MyAccount := reg
```

where reg is a register internal to the process (equivalent to a hardware register). We assume that each concurrent statement in the **parbegin** has its own register. That is, the parallel deposits expand into the primitives

Deposit 100	**Deposit 50**
reg1 := MyAccount;	reg2 := MyAccount;
reg1 := reg1 + 100;	reg2 := reg2 + 50;
MyAccount := reg1	MyAccount := reg2

The semantics of concurrent execution allows any possible permutation of primitive elements. One legal execution path is

Deposit100	**Deposit50**
reg1 := MyAccount;	•
MyAccount=1000 and reg1=1000	•
reg1 := reg1 + 100;	•
MyAccount=1000 and reg1=1100	•
•	reg2 := MyAccount;
•	*MyAccount=1000 and reg2=1000*
MyAccount := reg1	•
MyAccount=1100 and reg1=1100	•
•	reg2 := reg2 + 50;
•	*MyAccount=1100 and reg2=1050*
•	MyAccount := reg2
•	*MyAccount=1050 and reg2=1050*

Here the vertical axis represents the progression of time; we see the interleaving of concurrent primitives in the alternation of the statements. This execution

path shows that our program intended to deposit $150, but the value of the account has increased by only $50. The problem is that the bank account has been improperly shared. Each deposit function needs exclusive control of the bank account long enough to complete its transaction. Our program does not ensure this mutual exclusion.

In general, a *resource* in a computer system is something needed to complete a task. Resources can be physical objects, such as processors or peripherals, or software objects, such as memory locations, buffers, or files. Resources are shared, in that different concurrent activities can use them at different times, but resources are also private, in that there is an upper bound on the number of processes that can use a particular resource simultaneously. Often this bound is only a single process. For example, at some point many different processes may want to change the resource MyAccount. However, they should change it sequentially, one at a time. A concurrent activity is in the *critical region* of a resource when it is modifying or examining that resource. Thus, each of our processes is in a critical region of MyAccount when it executes the statement MyAccount := MyAccount + Deposit. The problem of preventing processes from executing simultaneously in critical regions over the same resource is the *mutual exclusion problem*. In general, processes that order their activities to communicate and not interfere with each other are *synchronized*. Synchronization includes aspects of cooperation as well as of serialized access to shared resources. Mutual exclusion is thus one facet of synchronization.

A process that is prevented from entering a critical region because another process is already in its critical region is *blocked*. If a set of processes are mutually blocked, that set is *deadlocked*. In a deadlocked set, no process can make further progress without external intervention. This occurs when several processes each need a resource to complete their tasks and the instances of that resource have been divided so as to preclude any process from getting enough to finish. For example, a system may have two tape drives, drive_a and drive_b. Both $\mathsf{process}_1$ and $\mathsf{process}_2$ require two drives (perhaps to copy tapes); $\mathsf{process}_1$ and $\mathsf{process}_2$ run concurrently. We imagine them to have the program schema shown in Figure 3-1. At statement (3), $\mathsf{process}_1$ is blocked; it needs $\mathsf{process}_2$ to release drive_b before it can continue. Similarly, $\mathsf{process}_2$ is blocked at statement (4) because $\mathsf{process}_1$ has possession of drive_a. The system halts, deadlocked.

Deadlock implies that all processes in a set are unable to accomplish useful work. A particular process *starves* if it is stuck, even though other processes continue to progress. We illustrate starvation by modifying our tape drive example to three processes and only a single drive. If after using the drive, $\mathsf{process}_1$ "passes" it to $\mathsf{process}_2$ and $\mathsf{process}_2$ back to $\mathsf{process}_1$, they can keep it to themselves, starving $\mathsf{process}_3$. In reality, starvation is more often the result of timing and priority relationships of programs. For example, $\mathsf{process}_3$ may starve if $\mathsf{process}_1$ and $\mathsf{process}_2$ both have a large appetite for the tape drive and the system gives higher priority to their requests.

	$\underline{\text{Process}_1}$	$\underline{\text{Process}_2}$
(1)	**request** drive_a	•
(2)	•	**request** drive_b
(3)	**request** drive_b	•
(4)	•	**request** drive_a
	the rest of process$_1$'s program	*the rest of process$_2$'s program*
(5)	**release** drive_a	•
(6)	•	**release** drive_b
(7)	**release** drive_b	•
(8)	•	**release** drive_a

Figure 3-1 Deadlocking resource demands.

3-2 SYNCHRONIZATION MECHANISMS

In 1968, Dijkstra published a classic paper on synchronization, "Co-operating Sequential Processes" [Dijkstra 68]. In that paper, he developed a software solution to the mutual exclusion problem. We follow his derivation in this section.

Dijkstra began by simplifying the general mutual exclusion problem to the mutual exclusion of two processes, process_1 and process_2. Process_1 and process_2 cycle through a loop; in each cycle each enters a critical region. Thus, the general schema for each process is

```
process_i:
    L_i:  critical region preparation;
          critical region;
          critical region cleanup;
          concurrent region;
          goto L_i
```

The mutual exclusion problem is to ensure that only one process is ever executing in its critical region at any time. The processes communicate by reading and writing shared variables. We assume that reading and writing a variable are each primitive (indivisible) operations.

A first attempt at a solution of this problem has the processes take turns entering their critical regions. We have a variable, turn, that indicates which process is to enter next. When a process wants to enter its critical region, it waits until turn has its process identifier.

```
var turn: integer;
begin
    turn := 1;
```

```
    parbegin
        process1:
            begin
                L1: if turn = 2 then goto L1;
                    critical region1
                    turn := 2;
                    concurrent region1
                    goto L1
            end;
        process2:
            begin
                L2: if turn = 1 then goto L2;
                    critical region2
                    turn := 1;
                    concurrent region2
                    goto L2
            end
    parend
end.
```

One way a blocked process can recognize an event is to execute a loop, looking at each iteration for the event. Such a process is said to be *busy waiting*, or *spinning*. Statements L1 and L2 implement busy waiting.

This solution ensures mutual exclusion—the two processes are never in their critical regions simultaneously. However, we have achieved mutual exclusion at the cost of synchronization—the processes are sure to enter critical regions in the order 1, 2, 1, 2, 1, 2, ... This synchronization is unpleasant; it should be possible to achieve more concurrency than this lockstep allows. The synchronization implied by this solution reduces the speed of each process to that of the slower (and if generalized to several processes would reduce the speed of every process to that of the slowest). To preclude such solutions, we impose another restriction on acceptable programs: "Stopping one process in its concurrent region does not lead to the other process blocking." We continue to assume that processes do not stop either in critical regions or in the preparation or cleanup for critical regions.

This leads us to the second solution. In this solution, we give each process a flag to show that it is in its critical region. To enter its critical region, a process checks if the other process is flying its flag, and, if not, raises its own and enters its critical region. The program to implement this algorithm is as follows:

```
var in_cr1, in_cr2: boolean;
begin
    in_cr1 := false;
    in_cr2 := false;
```

```
    parbegin
        process1:
            begin
                L1: if in_cr2 then goto L1;
                    in_cr1 := true;
                    critical region1
                    in_cr1 := false;
                    concurrent region1
                    goto L1
            end;
        process2:
            begin
                L2: if in_cr1 then goto L2;
                    in_cr2 := true;
                    critical region2
                    in_cr2 := false;
                    concurrent region2
                    goto L2
            end;
    parend
end.
```

This solution avoids the pitfall of synchronization. However, it does have a flaw—it fails to ensure mutual exclusion. Each process can find the other's flag lowered, raise its own, and enter its critical region. More specifically, the sequence

$\underline{\text{Process}_1}$	$\underline{\text{Process}_2}$
if in_cr2 **then** ...	•
•	**if** in_cr1 **then** ...
in_cr1 := true	•
•	in_cr2 := true
critical region$_1$	•
•	*critical region*$_2$

finds both processes simultaneously executing their critical regions.

Reversing the acts of checking the other process's flag and raising one's own ensures mutual exclusion. This gives us the following program:

```
var in_cr1, in_cr2: boolean;
begin
    in_cr1 := false;
    in_cr2 := false;
```

```
parbegin
    process1:
        begin
            L1:   in_cr1 := true;
            BW1:  if in_cr2 then goto BW1;
                  critical region1
                  in_cr1 := false;
                  concurrent region1
                  goto L1
        end;
    process2:
        begin
            L2:   in_cr2 := true;
            BW2:  if in_cr1 then goto BW2;
                  critical region2
                  in_cr2 := false;
                  concurrent region2
                  goto L2
        end;
parend
end.
```

This solution is safe. A process about to enter its critical region knows: (1) The other process's flag is down. Hence that process is not near its critical region. (2) Its own flag is flying. Hence the other process will not enter its critical region. Unfortunately, this program is susceptible to deadlock. Each process raises its flag and then loops, waiting for the other to lower its flag. Temporally, this is

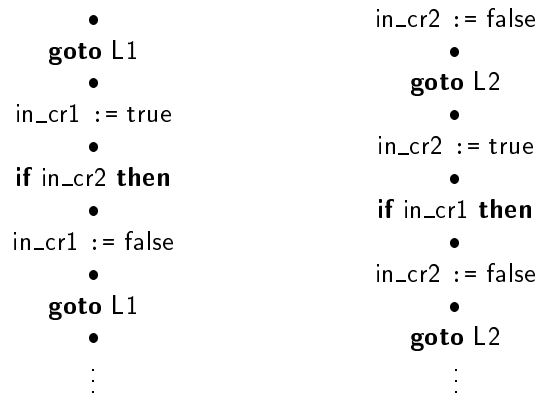
$\underline{\text{Process}_1}$	$\underline{\text{Process}_2}$
in_cr1 := true	•
•	in_cr2 := true
BW1: **if** in_cr2 **then goto** BW1	•
•	BW2: **if** in_cr1 **then goto** BW2
BW1: **if** in_cr2 **then goto** BW1	•
•	BW2: **if** in_cr1 **then goto** BW2
BW1: **if** in_cr2 **then goto** BW1	•
•	BW2: **if** in_cr1 **then goto** BW2
⋮	⋮

The problem is that the processes have been too stubborn about keeping their flags flying. If a process cannot enter its critical region, it needs to back off, lowering its flag before trying again. This brings us to our penultimate program:

```
var in_cr1, in_cr2: boolean;
begin
    in_cr1 := false;
    in_cr2 := false;
    parbegin
        process1:
            begin
                L1: in_cr1 := true;
                    if in_cr2 then
                    begin
                        in_cr1 := false;
                        goto L1
                    end;
                    critical region1
                    in_cr1 := false;
                    concurrent region1
                    goto L1
            end;
        process2:
            begin
                L2: in_cr2 := true;
                    if in_cr1 then
                    begin
                        in_cr2 := false;
                        goto L2;
                    end;
                    critical region2
                    in_cr2 := false;
                    concurrent region2
                    goto L2
            end;
    parend
end.
```

This solution *almost* works. By lowering its flag, each process gives the other a chance to succeed. However, it is possible that the two processes might follow the sequence

$\underline{\text{Process}_1}$	$\underline{\text{Process}_2}$
in_cr1 := true	•
•	in_cr2 := true
if in_cr2 **then**	•
•	**if** in_cr1 **then**
in_cr1 := false	•



Is such synchronization possible? In systems composed of identical elements, it may even be likely, so we must reject this solution too. This situation, in which a system's processes are not blocked but still fail to progress, is called *livelock*.

Given these constraints, a correct solution to the mutual exclusion problem is surprisingly difficult to program. Thomas Dekker proposed the first such solution. His solution combines elements of the turn variable of the first program with the flags of the later programs. The key idea is to detect potential "after you" situations (when a process has its own flag raised, and sees the other process's flag) and to resolve them by giving priority to the process marked by the turn variable. After each critical region, a process sets the turn variable to the identity of the other process, giving the other priority if there is a conflict the next time around. Unlike the last solution, the processes ignore this priority if there is no conflict.

```
var in_cr1, in_cr2: boolean;
    turn: integer;
begin
    turn   := 1;
    in_cr1 := false;
    in_cr2 := false;
    parbegin
        process1:
            begin
                L1:   in_cr1 := true;
                PW1: if in_cr2 then
                        begin
                            if turn = 1 then goto PW1;
                            in_cr1 := false;
                            BW1: if turn = 2 then goto BW1;
                            goto L1
                        end;
```

```
                    critical region1
                    turn := 2;    -- give the other process priority
                    in_cr1 := false;
                    concurrent region1
                    goto L1
            end;
        process2:
            begin
                L2:   in_cr2 := true;
                PW2: if in_cr1 then
                    begin
                        if turn = 2 then goto PW2;
                        in_cr2 := false;
                        BW2: if turn = 1 then goto BW2;
                        goto L2;
                end;
                critical region2
                turn := 1;
                in_cr2 := false;
                concurrent region2
                goto L2
            keywordend;
    parend
end.
```

This progression ought to leave the reader with some feeling for the additional complications inherent in programming concurrent systems. We must not only ensure that a particular segment of program is correct, but also that no concurrent activity can jeopardize it. Dekker's solution solves the problem, but the solution itself is complicated and unappealing. Because it is too cumbersome for practical purposes, the search began for alternative mechanisms.*

One reason the problem is difficult is that the actions of reading and writing shared storage are not indivisible primitives. This led to the idea of an instruction that would both read and write storage as a single action. The TS (test-and-set) operation on the IBM/360 was an early implemention of this idea. The instruction both stored a value in a memory location and returned the previous value from that location. To processes running on the machine, the instruction appeared indivisible.

Solutions using test-and-set have the disadvantage of requiring a process waiting for a resource to loop, checking to see when the resource is free. Instead, it would be better to have a primitive that would combine the precise intention

* Peterson has demonstrated a simpler solution of the mutual exclusion problem. We present his algorithm in Chapter 6.

of both protecting a resource and waking a waiting process. In "Co-operating Sequential Processes," Dijkstra introduced the semaphore for just this purpose [Dijkstra 68]. A *semaphore* is an (abstract) object with two operations, one to claim the resource associated with the semaphore and the other to release it. The claiming operation on a semaphore s is **P**; the releasing operation is **V**. If the semaphore claimed is busy, the requesting process is blocked until it is free. When another process executes the **V** on that semaphore, a blocked process is released and allowed access to the resource.

Semaphores are typically implemented as nonnegative integers. Executing **V**(s) increments s; executing **P**(s) tries to decrement s. If s is positive, this action succeeds; if it is zero, then **P** waits until it is positive. Instead of a busy wait, the blocked process can be added to a set of processes waiting on that semaphore. These incrementing and decrementing operations must be indivisible. The initial value of s is the number of processes that can simultaneously access the resource. A *binary semaphore* allows only one process to control the resource at any time. A semaphore variable whose initial value is one acts as a binary semaphore. A program for mutual exclusion using semaphores is as follows:

```
var mutex: semaphore;
begin
    mutex := 1;
    parbegin
        process1:
            begin
                L1: P(mutex);
                    critical region1
                    V(mutex);
                    concurrent region1
                    goto L1
            end;
        process2:
            begin
                L2: P(mutex);
                    critical region2
                    V(mutex);
                    concurrent region2
                    goto L2
            end;
    parend
end.
```

This solution has the additional advantage of being able to enforce the mutual exclusion of any number of processes without modification.

Semaphores were a conceptual advance over setting and testing shared memory but failed to provide additional structuring to exclusion and sharing. Other

proposed primitives for resource control include locks [Dennis 66] and monitors [Hoare 74]. Locks abstract the simple idea of a resource "lock" with two operations, lock and unlock. A process trying to lock an already locked lock spins until it is unlocked. Monitors combine both data abstraction and mutual exclusion into a single sharable object; access to that object is not only serialized but is also abstract. In Section 13-1, we discuss Concurrent Pascal, a language that uses monitors.

Synchronization mechanisms like semaphores embody the notion of indivisible, primitive actions. Often an action should seem to be indivisible, but is more complex than a single primitive. An *atomic action* is a compound computation that is distributed over time or location, but cannot be externally decomposed into more primitive actions. Following Leslie Lamport, we use angle brackets ($\prec\succ$) to delimit atomic actions [Lamport 80]. Thus, the notation

$$\prec A;\ B;\ C\succ$$

indicates that A, B, and C are to be executed sequentially and atomically.

3-3 ILLUSTRATIVE EXAMPLES

Earlier in this chapter we discussed the issues involved in concurrent access to a bank account variable. We did not select this example as an illustration of what to do if you ever find yourself programming for Chase Manhattan. Instead, the concept of a shared resource with state that can be tested and set is a common theme in programming concurrent systems. The metaphor of a bank account is just an instance of this theme.

Several such "metaphorical" examples have been proposed that summarize particular problems associated with resource control and concurrency. We use five of these as illustrative examples in Parts 2, 3, and 4. We have already described two of them, shared storage and semaphores. We call the shared storage problem the register problem. *Registers* have two operations, one that stores a value in the register and another that returns the last value stored. Of course, semaphores also have two operations, **P** and **V**. A process executing **P** on a depleted semaphore is blocked or refused until another process replenishes that semaphore with a **V**.

Our three other standard examples are the readers-writers, dining philosophers, and producer-consumer buffer problems.

Readers and writers Courtois, Heymans, and Parnas described the readers-writers problem in 1971 [Courtois 71]. This problem illustrates a variety of mutual exclusion that is more complex than simple semaphorelike mutual exclusion but nevertheless realistic. The readers-writers problem posits two classes of users of a resource, readers and writers. Readers *read* the resource and writers *write* it. Readers can share access to the resource. That is, several readers can be reading

the resource simultaneously. Writers require exclusive control of the resource. When a writer is writing, no other process can be reading or writing.

The multiple-readers/single-writer pattern of resource control corresponds to a desirable way of accessing shared databases. This pattern allows many processes to be simultaneously reading the database, but restricts database updates to a single process at a time. Many database update transactions involve making several changes in the database. A process that reads the database during an update might obtain an inconsistent view of the data.*

Solutions of the readers-writers problem should maximize concurrent access to the database without starving either readers or writers. Solutions usually allow readers to read until a writer wants to write. Additional readers are then blocked. When the currently reading readers have finished, the system allows the writer to write and then unblocks the waiting readers. The process is then repeated.

Dining philosophers The bank account example illustrated mutual exclusion of two processes and a single resource. Of course, computing systems can have many processes and many resources, with many different patterns of resource exclusion. The readers-writers problem is an example of one such pattern. Dijkstra's dining philosophers problem describes another, more fanciful, exclusion regime [Dijkstra 72a]. Five philosophers live an existence centered around two activities: thinking and eating. In a room is a circular table and in the middle of the table, a serving platter of spaghetti. Each philosopher has her own place at the table; at that place is a plate. A fork lies between each pair of plates. A philosopher's life is a simple one. She thinks. Becoming hungry, she enters the room and takes her place at the table. Since eating spaghetti requires two forks, she picks up first one fork and then the other. She then fills her plate with spaghetti and eats. Satiated, she returns the forks to their places, leaves the room, and resumes thinking, eventually repeating the cycle. Figure 3-2 shows the dining arrangement of the philosophers.

The reader should understand that these philosophers are stubborn characters. Once having become hungry, entered the room, and acquired a fork, they do not relinquish that fork until after they have eaten. And, of course, self-respecting philosophers would rather starve than eat with only one fork.

Given these rules, all five philosophers can become hungry at roughly the same time, enter the room, sit, pick up one fork (say, their left forks), and wait, interminably, for the other fork to become free. At this point, the philosophers are deadlocked and (literally) starve.

A solution to the dining philosophers problem is a program that models this situation. The philosophers are usually modeled as processes. Processes or data structures may be used to model the behavior of the forks and the dining room. Each philosopher should have virtually the same program, differing only in the

* We discuss control algorithms for database systems in Chapter 17.

Figure 3-2 The dining philosophers.

philosopher's seat and fork assignments. A deadlock-free solution to the dining philosophers problem is one in which some philosopher eventually eats — that is, some work gets done. A starvation-free solution is one in which no philosopher starves if she waits long enough to obtain forks. Of course, a deadlock-free solution is a more complex program than the simple modeling problem and a starvation-free solution is still more complex.*

The dining philosophers problem reflects the common need of real systems for multiple resources to accomplish their tasks. If five programs share five tape drives and each needs two drives, then an organization that gives one tape drive to each program can starve or deadlock the computer system as easily as poor dining room management can starve philosophers.

Buffers Often two processes are coupled in a "producer-consumer" relationship: the producer process generates data that is used by the consumer process. For

* Variations on the dining philosopher's problem include generalizing the number of philosophers, changing the possible actions of a hungry, forkless philosopher, and, more whimsically, substituting chopsticks and Chinese food for forks and spaghetti.

example, a producer process might generate output to be printed, while a line printer (consumer) process might take that output and drive the printer to write it.

Clearly, two such processes can be organized procedurally. When the producer has a ready datum it can call the consumer, as a procedure, to handle it. The consumer can return an acknowledgment to the producer when it finishes processing. (Or, conversely, the consumer could call the producer for the next data item.) However, this architecture leaves the producer idle while the consumer computes and the consumer idle while the producer computes. That is, this organization yields no concurrency.

We can obtain concurrency by using an (unbounded) buffer. When the producer has a ready datum, it sends it to the buffer. When the consumer wants the next datum, it asks the buffer for it. If the buffer is empty, the buffer either delays the consumer or informs it of the lack of data. The producer is never slowed; it can always be generating data and adding it to the buffer. The buffer acts as a queue of unconsumed data.

The producer-consumer buffer problem can be generalized to multiple producers and consumers. For example, a system may have several producer processes that intermittently create output for printing and several printer processes capable of consuming that output and creating listings. Producers do not care which printer prints their output. A buffer between the producers and the consumers allows not only the concurrency of data generation and printing but also the delegation of printing tasks independent of printer identity.

We have described an unbounded producer-consumer buffer, where producers never need to be concerned about the availability of buffer space for their output. The bounded producer-consumer problem assumes that the buffer can hold only a fixed number of data items. Solving the bounded buffer problem requires not only an appropriate response to a consumer when the buffer is empty but also an appropriate response to a producer when the buffer is full. A producer that tries to insert an element in a full buffer must be either rejected or delayed until space becomes available.

PROBLEMS

3-1 Generalize Dekker's solution to the mutual exclusion problem from 2 processes to n processes.

3-2 Dekker's solution precludes deadlock. Does it also preclude starvation?

3-3 In Dekker's solution, what happens to the system if a process fails while executing in its critical region? What happens if it fails while executing the code for the critical region preparation or cleanup?

3-4 Program mutual exclusion using the test-and-set operation.

3-5 Program the readers-writers problem using semaphores.

3-6 Program the dining philosophers problem using semaphores.

3-7 Program a bounded producer-consumer buffer using semaphores.

REFERENCES

[**Courtois 71**] Courtois, P. J., F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers,'" *CACM*, vol. 14, no. 10 (October 1971), pp. 667–668. Courtois et al. describe and solve the readers-writers problem. Their solution uses semaphores.

[**Dennis 66**] Dennis, J. B., and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *CACM*, vol. 9, no. 3 (March 1966), pp. 143–155. Dennis and Van Horn propose the mutual exclusion mechanism of explicit primitives that lock and unlock a variable.

[**Dijkstra 68**] Dijkstra, E. W., "Co-operating Sequential Processes," in F. Genuys (ed.), *Programming Languages: NATO Advanced Study Institute*, Academic Press, London (1968), pp. 43–112. This paper is an excellent exposition on the difficulties of concurrent programming. Starting from simple ideas about concurrency, Dijkstra develops the ideas of Dekker's mutual exclusion algorithm and semaphores. He then generalizes the mutual exclusion problem to multiple processes.

[**Dijkstra 72a**] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," in C.A.R. Hoare, and R. H. Perrott (eds.), *Operating Systems Techniques*, Academic Press, New York (1972), pp. 72–93. This paper is an introduction to mutual exclusion, including a description of the dining philosophers problem.

[**Hoare 74**] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *CACM*, vol. 17, no. 10 (October 1974), pp. 549–557. Hoare describes the monitor concept and argues for its value in programming operating systems. We discuss Concurrent Pascal, a language based on a simpler type of monitor, in Section 13-1.

[**Lamport 80**] Lamport, L., "The 'Hoare Logic' of Concurrent Programs," *Acta Informa.*, vol. 14, no. 1 (1980), pp. 21–37. Lamport introduces "$\prec\succ$" pairs to indicate atomic actions.

[**Shaw 74**] Shaw, A. C., *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey (1974). Shaw develops the issues of concurrency and mutual exclusion within the context of building operating systems. Many of the issues of operating system development apply to coordinated computing.

[**Stark 82**] Stark, E. W., "Semaphore Primitives and Starvation-Free Mutual Exclusion," *JACM*, vol. 29, no. 4 (October 1982), pp. 1049–1072. Stark identifies three different implementation techniques for semaphores: (1) queueing processes blocked on a semaphore, (2) keeping processes blocked on a semaphore in a "blocked set," selecting the next process to be released on a **V** operation from that set, and (3) letting blocked processes spin, with the first to notice that the semaphore has been released being the one that claims it. Stark shows that the first two techniques are more powerful than the third, in that "good" starvation-free mutual exclusion can only be programmed with blocked sets or queues.